A desirable algorithm: maximum weight scheduling

→ Throughput optimal for essentially any network model.
→ For the 'correct' choice of weight, it induces low delay.

Question: design an algorithm to find

\[ \sigma(t) \in \arg\max_{\sigma \in \mathcal{S}} \sum_{i=1}^{n} \sigma_i \cdot W_i(t), \]

where \( \mathcal{S} \) is the set of schedules and
\( W_i(t) \) is the weight of queue \( i \) at time \( t \).

Example: For input-queued switch, of \( n \)-ports,
the set of schedules, \( \mathcal{S} \) is equal to the set of
matchings in \( n \times n \) complete bipartite graph.

That is,

\[ \mathcal{S} = \left\{ \Pi = [\pi_{ij}] \in \{0,1\}^n : \sum_{k=1}^{n} \pi_{ik} = 1, \sum_{k=1}^{n} \pi_{kj} = 1, \right\} \]

for all \( 1 \leq i,j \leq n \)

Question of scheduling: find \( \Pi(t) \), which is a
solution of

\[ \max \sum_{i,j=1}^{n} \pi_{ij} w_j(t) \]

over \( \Pi \in \{0,1\}^n \)

\[ \sum_{k=1}^{n} \pi_{ik} = 1, \sum_{k=1}^{n} \pi_{kj} = 1, \forall i,j. \]

A general method: Dynamic programming.

Definition: \( P(\mathcal{S},W,n) \equiv \max \sum \sigma_i \cdot W_i \) over \( \mathcal{S} \).

Compute \( P(\mathcal{S},W,n) \)

If \( n=1 \), then return \( \max \{0,W_i\} \).

Else

\[ P(\mathcal{S},W,n) = \max \left\{ W_i + P(\mathcal{S}(\mathcal{S}_{\leq i}),W_{\leq i}); P(\mathcal{S}(\mathcal{S}_{> i}),W_{> i}) \right\} \]

In above,
\( \mathcal{S}(\mathcal{S}_{\leq i}) = \{ \sigma \in \mathcal{S} : \sigma_i = 1 \} \)
\( \mathcal{S}(\mathcal{S}_{> i}) = \{ \sigma \in \mathcal{S} : \sigma_i = 0 \} \).
Approximate dynamic programming

The precise dynamic programming can be very expensive, computationally. For example, naive implementation can cost \( \exp(\Theta(n^2)) \) operations. Therefore, such algorithms is “impossible” to be implementable.

Basic question: what is implementable?

1. Distributed algorithm: each “node” performs its own computation.
2. Few logical operations per node.
3. Iterative design \( \rightarrow \) pipelineable.
4. Little data structure

We wish to design such algorithm that has good performance. An approach we shall take is that based on approximate dynamic programming. This approach is structural.

First, the approach based on “Tree” for matching problem. Later, we shall describe history of the approach and known properties followed by its implementation in the context of a switch.

Tree-based approximation through an example.

find max. weight matching in the tree graph \( G = (V, E) \) with \( V \) edge weights

\[ W = [w_{ij}]_{(i,j) \in E} \]
Use dynamic programming for finding to which node A should connect to or not connect to under the max. weight matching. That is, first consider options:

- A connects to B
- A connects to C
- A connects to none.

Under A connects to B:

```
   A
  /|
 / | \
B -- C
  \
   D E F G
```

On left tree, B is occupied, and so for that subtree, B can not connect to D or E.

Given that, D has to take ‘best option’, assuming no connection from B. That is, D connects to none in this case.

That is, D will send ‘information’ to B ‘0’.

More generally, D has to send ‘two-pieces’ of information to B:

\[
\begin{align*}
m_{D\rightarrow B} &= \text{cost of solution from D's perspective of connecting D \& B} \\
&= W_{BD} \quad (= 10) .
\end{align*}
\]

\[
\begin{align*}
m_{D\rightarrow B} &= \text{cost of solution from D's perspective of not connecting D \& B} \\
&= 0 \quad (=0) .
\end{align*}
\]

Similarly, \( m_{E\rightarrow B} = W_{BE} \)

\[
\begin{align*}
m_{E\rightarrow B} &= 0
\end{align*}
\]
Using these we wish to compute: \( m_{B \to A}, m_{B \to A} \).

\( m_{B \to A} : \) if \( B \) connects to \( A \) then

\[
0 \quad \text{D and E cannot connect to B.}
\]

Therefore,

\[
m_{B \to A} = W_{AB} + m_{D \to B} + m_{E \to B} \tag{1}
\]

Similarly,

\( m_{B \not\to A} : \) if \( B \) does not connect to \( A \) then

the best option is the one with max. cost among \( D \to B, E \to B \) or none.

Therefore,

\[
m_{B \not\to A} = 0 + \max \left\{ m_{D \to B} + m_{E \to B}, m_{D \not\to B} + m_{E \not\to B} \right\} \tag{2}
\]

\((1) - (2) \Rightarrow \)

\[
m_{B \to A} - m_{B \not\to A} = W_{AB} - \max \left\{ m_{D \to B} - m_{D \not\to B}, m_{E \to B} - m_{E \not\to B} \right\}
\]

Define: \( M_{i \to j} \equiv m_{i \to j} - m_{i \not\to j} \)

Recursion:

\[
M_{i \to j} = W_{i \to j} - \max \left\{ B_{i \to j}, 0 \right\}
\]
Approximate algorithm: Belief Propagation.

0. Variables:
   - iteration index: \( t \in \{0, 1, 2, \ldots \} \)
   - for each \((i, j) \in E\), iteration \( t \)
     \[
     B_{i \to j}^t, B_{j \to i}^t
     \]

1. Initially, \( t = 0 \) and \( B_{i \to j}^0 = B_{j \to i}^0 = W_{ij} \).

2. Update
   \[
   B_{i \to j}^{t+1} = W_{ij} - \max_{\ell \in N(i) \setminus \{j\}} \left\{ B_{\ell \to i}^t, 0 \right\}
   \]

3. Estimate:
   \[
   \Pi_{i}^{t+1} = \begin{cases} 
   k & \text{if } \left( \max_{\ell \in N(i)} B_{\ell \to i}^{t+1} \right) = B_{k \to i}^{t+1} > 0 \\
   \emptyset & \text{otherwise}.
   \end{cases}
   \]

4. Set \( t = t + 1 \) and repeat till convergence of estimates.
Implementation Concerns.

One. Does algorithm converge?

Is the solution correct?

Two. Can we simplify algorithm further at loss of performance?

Finally. How does it compare to popular implemented solution?

One. Performance of Belief Propagation.

Theorem. Consider nxn complete bipartite graph G with edge weight \( W = \{ w_{ij} \}_{nxn} \) such that (a) \( w_{ij} \geq 0 \) and (b) there is a unique maximum weight matching. Then, algorithm finds optimal (max. wt. matching) after

\[ t = \left\lceil \frac{2nW^*}{\varepsilon} \right\rceil \] iterations.

\[ E = \text{wt. of max. wt. matching} - \text{wt. of 2nd max. wt. matching} \]

\[ W^* = \max_{i,j} w_{ij} \]

Total operations:

(*) 2 messages per-edge per iteration and hence \( 2n^2 \) messages.

\( \rightarrow \) Each node can compute it's n messages in \( 2n \) operations.

\( \rightarrow \) Total \( O(n^2) \) operations per iteration.

(*) Total: \( O(n^3) \) operations.
Two. Implementation of BP.

For convergence, we need unique solutions.

Further, convergence time depends on the problem parameter.

We need an algorithm that always converges in the pre-determined number of steps.

Solution: Randomization.

Set: $D = \frac{E(W^*)}{n}$ for some $\varepsilon > 0$.

Set: $\tilde{W}_{ij} = W_{ij} + D_{ij}$, where

$D_{ij}$ is chosen uniformly at random from $[0, D]$.  

Theorem. BP algorithm convergence in $O(n^{1/\varepsilon})$ iterations with a solution being at least $(1-\varepsilon)$ approximate.

This should imply $(1-\varepsilon)$ 100% throughput optimal.

This suggests trade-off between cost and performance.
Finally, stringent implementation constraints.

1. Limited number of bits allowed.

   - \( \left\lceil \log \log W \right\rceil + L \) bits required to find weight \( \tilde{W} \) s.t. \( \tilde{W} \equiv (1 - \frac{1}{2^L}) \cdot W \).
   - And, hence weight approximation is \( \sim (1 - \frac{1}{2^L}) \).

2. Only one-bit allowed.

   - \( W_{ij} = 0 \) or \( > 0 \)
     - i.e. queue empty or not.

   Then, algorithm should be:

   - left parent
     - Each node \( i \) creates preference of nodes \( j \) s.t. \( W_{ij} > 0 \).
   - Similarly, each \( j \) creates preference of \( i \)'s s.t. \( W_{ij} > 0 \).

Use these preferences to come up with a maximal matching.

Use "stable marriage" algorithm for this.

This is called ESUP. If preferences are "round-robin" over time.